УДК 372.800.442
ББК Ч426.32-26+3973р ГСНТИ 14.35.09 Код ВАК 13.00.02

Емельянов Дмитрий Александрович,
кандидат технических наук, доцент кафедры информатики, информационных технологий и методики обучения информатики, Институт математики, информатики и информационных технологий, Уральский государственный педагогический университет; 620075, г. Екатеринбург, ул. К. Либкнехта, 9; e-mail: eda@uspu.ru.

ОСВОЕНИЕ ПРИНЦИПОВ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ В ХОДЕ РАЗРАБОТКИ ИГРОВЫХ ПРОГРАММ

КЛЮЧЕВЫЕ СЛОВА: объектно-ориентированное программирование, среда визуальноного программирования Delphi, игровые программы.

АННОТАЦИЯ. В статье рассмотрено использование в качестве учебного материала при обучению объектно-ориентированному программированию в учебных заведениях задач, ориентированных на разработку игровых программ. В качестве примера приведены общие подходы к созданию приложения игра «Шахматы», в котором продемонстрировано применение современной методики объектно-ориентированного программирования. В процессе демонстрационного разбора предлагаемого приложения, в ходе написания программного кода показано использование основных принципов объектно-ориентированного программирования: инкапсуляции, наследования и полиморфизма. В качестве среды разработки выбрана среда визуальноного программирования Delphi. Это связано с тем, что данный продукт широко распространен при обучении программированию в современных учебных заведениях, а сама игра является замечательным примером, к которому может быть применен именно объектно-ориентированный подход. В настоящее время подавляющее большинство современных информационных продуктов в мире (в том числе и для мобильных устройств) реализуются с помощью технологии объекно-ориентированного программирования, именно поэтому изучение и овладение этим материалом является сегодня весьма актуальной задачей. Для быстрого освоения технологии программирования в среде Delphi можно воспользоваться учебными методическими пособиями, предложенными автором данной статьи.

Emel'yanov Dmitry Alexandrovich,
Candidate of Technical Sciences, Associate Professor of Department of Informatics, Computer Technology and Methods of Teaching Informatics, Institute of Informatics and Information Technologies, Ural State Pedagogical University, Ekaterinburg.

STUDY OF PRINCIPLES OF THE OBJECT-ORIENTED PROGRAMMING IN CREATION OF GAME PROGRAMS

KEYWORDS: object oriented programming, Rapid Application Development Delphi, programming games.

ABSTRACT. This article is the illustration of basic principles of object-oriented programming for creation of game applications. As an example, the main approaches to the creation of the game 'Chess' in RAD (Rapid Application Development) Delphi is given, which shows the use of the modern methods of object-oriented programming. While showing the application and creation of the program code, the author demonstrated the main principles of object-oriented programming: encapsulation, inheritance and polymorphism. Rapid Application Development Delphi is chosen as the basis, baceause it is very popular in modern school education and the game 'Chess' is a wonderful example to apply object-oriented method. Today object-oriented technology (Object Oriented Programming - OOP) is the realization of the most modern information products in the world (including mobile gadgets), this makes knowledge of this technology very important. The author suggests his own educational manuals for rapid mastering of object-oriented programming in RAD Delphi.

Современные программные продукты разрабатываются на базе передовых технологий, одним из которых является объектно-ориенированный метод [1–4; 15]. При изучении программирования желательно использовать такой учебный материал, который будет вызывать непосредственный интерес у обучаемых. Для этой цели очень хорошо использовать примеры, иллюстрирующие разработку игровых программ, которые вызывают повышенный интерес у обучаемых. Такая мотивация очень хорошо зарекомендовала себя в реальном учебном процессе. В данной статье рассмотрены общие подходы к разработке игровых программ с использованием объектно-ориентированного подхода на примере создания приложения «Шахматы» [8].

Объектно-ориентированное программирование (ООП) – современная технология программирования, позволяющая разрабатывать сложные системы, устойчивые по отношению к ошибкам и допускающие последующее расширение. ООП аккумулирует лучшие идеи, воплощенные в структурном программировании [9–11] и сочетает их с мощными концепциями, которые позволяют оптимально организовать программы [5; 12; 14]. ООП позволяет разло-

жить проблему на составные части – каждая составляющая становится самостоятельным объектом, содержащим свои собственные коды и данные, относящиеся к этому объекту. Программа в целом представляет собой набор объектов и связей между ними.

ООП базируется на типе данных, называемым «Класс» (Class), и поддерживает три основные концепции: инкапсуляция, наследование и полиморфизм.

Постановка задачи

Игра «Шахматы» дает возможность прекрасно проиллюстрировать возможности современного программирования с использованием объектно-ориентированного подхода. Перед тем как начать создавать программу, сформулируем основные требования, которые необходимо выполнить. Цели и задачи нашей программы будут заключаться в том, чтобы обепечить:

- создание шахматной доски с буквенно-цифровой разметкой;
- начальную расстановку шахматных фигур на доске;
- реализацию ходов всех шахматных фигур по правилам;
- возможность «срубить» фигуру по правилам;
- счет количества ходов и их очередность;
- обработку ситуаций «ШАХ», «МАТ» и «ПАТ».

Таким образом, эта программа не будет предназначена для игры пользователя с компьютером, то есть в нашу задачу не входит разработка каких-либо алгоритмов обработки шахматной ситуации. С помощью подобной программы можно будет просто играть двум соперникам, пользуясь собственными практическими навыками шахматной игры. Реализация программы может быть самой различной в зависимости от вкуса и желания программиста, поэтому в данной статье постараемся описать проблему в самом общем виде, при этом рассмотрим несколько возможных вариантов ее решения.

Последовательность разработки программы

Шаг первый – создание шахматной доски

В первую очередь, создадим шахматную доску, по которой будут перемещаться фигуры. Прежде всего необходимо решить, из чего будет состоять доска, так как это будет существенно влиять не только на то, как мы ее будем отображать, но и как будут перемещаться по ней фигуры. Для конечного пользователя это не будет иметь абсолютно никакого значения, а вот для программиста это является главным и весьма существенным фактором. Кратко опишем два возможных варианта создания шахматной доски, используя компоненты *TPanel* и *TShape*.

Определим размеры окна будущего приложения, которые следует поместить в обработчик события *OnCreate* главной формы:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    ClientHeight:=440;
    ClientWidth:=440;
    BorderStyle:=bsSingle;
end;
```

Здесь первые две строки отвечают за высоту и ширину клиентской части формы (окна приложения) соответственно, а третья строка запрещает изменять размеры формы. Значение 440 получилось следующим образом: 20 px — расстояние от всех краев формы до доски, где будет нанесена буквенно-цифровая разметка; 8 клеток по 50 px дадут нам еще 400 px. Итого, получим значение в 440 px.

Доска как экземпляр класса TShape или TPanel

В разделе объявления глобальных переменных *VAR* или в разделе *private* главной формы (все зависит от конкретной реализации) объявим двумерный массив переменных 8 на 8 типа *TKletka* следующим образом:

```
DOSKA: array[1..8,1..8] of TKletka;
```

В дальнейшем, массив клеток доски будем обозначать именно так для обобщения всех вариантов конструирования доски. Здесь в качестве объектов типа *TKletka* (запись в общем виде) в конкретной реализации программы можно использовать объекты типа *TShape* (массив фигур-квадратов) или объекты типа *TPanel* (массив панелей).

Отрисовка доски будет происходить также при событии *OnCreate* главной формы, куда следует поместить следующий код:

```
//---------- Создание шахматной доски ----------
for i:=1 to 8 do //Двойной цикл обеспечит нам квадрат из клеток 8 на 8
 for j:=1 to 8 do
 begin
 DOSKA[i,j]:=TKletka.Create(Form1);
//Вызов конструктора класса для выделения памяти под объект
 DOSKA[i,j].Parent:=Form1;
//Определяем объект-родитель, в который будут помещаться клетки доски:
//поскольку размещение на главной форме, ее и указываем
 DOSKA[i,j].Height:=50;
 DOSKA[i,j].Width:=50;
//Эти две строки отвечают за размеры клеток: высота и ширина
 DOSKA[i,j].Left:=20+50*(i-1);
 DOSKA[i,j].Top:=20+50*(j-1);
//Здесь определяется положение клеток:
//Left – расстояние от левого края формы до объекта,
```

```
//Top - от верхнего края формы до объекта.
 if odd(i+j) then DOSKA[i,j].Цвет:=clSilver
 else DOSKA[i,j].Цвет:=clWhite
 DOSKA[i,j].OnDragDrop:=
Form1.OnImageDrop;
 DOSKA[i,j].OnDragOver:=
Form1.OnImageOver;
 end;
```

Все, доска готова. Можно запустить программу и посмотреть, корректно ли она отобразилась. При запуске следует закомментировать две строчки событий Drag&Drop, поскольку они пока еще не реализованы. Если все набрано правильно, то на форме отобразится доска 8 на 8, причем левый нижний квадрат должен быть черным.

Для полной картины следует подписать клетки буквами от A до H по горизотнали и цифрами от 1 до 8 по вертикали. Для этого в разделе объявления глобальных переменных или в разделе *private* главной формы объявим двумерный массив меток:

```
POLE: array[1..4,1..8] of TLabel;
```

Таким образом, мы объявили переменные для 32-х меток – 16 из них для букв сверху и снизу доски, а другие 16 – для цифр слева и справа. Ниже приведен еще один фрагмент кода события *OnCreate* главной формы, который позволит сделать на доске необходимые надписи:

```
for i:=1 to 4 do
for j:=1 to 8 do
 begin
 POLE[i,j]:=TLabel.Create(Form1);      //Создание
метки
 POLE[i,j].Parent:=Form1;      //на      форме
(родитель)
 POLE[i,j].Font.Style:=[fsBold]; //Стиль шрифта
 POLE[i,j].Transparent:=true;      //Прозрачная
метка
 with POLE[i,j] do
 case i of
 1: begin Left:=40+(j-1)*50; Top:=5;
 Caption:=chr(64+j); end; //Верхняя строка (A-
H)
 2: begin Left:=5; Top:=Form1.ClientHeight-j*50;
 Caption:=IntToStr(j); end; //Левый столбец (1-8)
 3: begin Left:=40+(j-1)*50; Top:=425;
 Caption:=chr(64+j); end; //Нижняя строка (A-H)
 4: begin Left:=425; Top:=Form1.ClientHeight-
j*50;
 Caption:=IntToStr(j); end; //Правый столбец (1-8)
 end; {Case}
 end;
```

Теперь можно запустить программу еще раз и проверить, как выглядит общий вид шахматной доски в окончательном виде с нанесенной разметкой.

Шаг второй – проектирование класса TFigura

На данном этапе нашей задачей является правильно спроектировать класс *TFigura*. Для этого определим основные характеристики шахматных фигур и операции над ними, которые нам будет необходимо реализовать в проекте.

Во-первых, сначала необходимо выбрать «класс-родитель», чтобы наследовать максимум необходимых характеристик (сущностей) и методов. Так как доска реализована объектами типа *TShape* или *TPanel*, на которые будут помещаться изображения шахматных фигур, то в этом случае в качестве родительского класса следует выбрать класс *TImage*.

Во-вторых, чтобы отобразить фигуры на экране, расставить их на доске и присвоить всем полям начальные значения, нам необходим *конструктор*. Его реализация рассмотрена далее.

В-третьих, необходимо хранить координаты фигур (начальные – до совершения хода и конечные – после совершения хода), которые нужно реализовать свойствами. Следовательно, должны присутствовать методы записи и чтения в эти свойства непосредственно из полей. Поскольку в шахматах имеется только два цвета фигур (белые и черные) его лучше реализовать не в виде свойства, а в виде логической переменной, которая принимает значение *true* для белых фигур и *false* для черных. Хотя по желанию программиста цвет фигур можно также оформить в виде свойства.

В-четвертых, все фигуры должны ходить по правилам, следовательно, должен существовать метод, определяющий правило хода (HOD) для каждой фигуры. Так как в шахматах существует 6 разновидностей фигур (Король, Пешка, Ферзь, Слон, Конь и Ладья), нужно объявить у нашего класса *TFigura* шесть классов-потомков (подклассов) и сделать метод правила хода *полиморфным*.

В-пятых, чтобы фигуры не могли прыгать друг через друга, необходим метод, выполняющий соответствующую проверку. Назовем его *JUMP* – метод прыжка. Здесь исключение составляют Конь (ему разрешено прыгать через фигуры) и Король (он вообще «не умеет прыгать») – его перемещение в любую сторону состаляет только одну клетку.

В-шестых, у фигур на основании двух предыдущих методов (ход по правилам с учетом занятых клеток на пути фигуры) должен присутствовать метод перемещения на новое место – *правильный ход*. Он, в свою очередь, может быть реализован двумя разными способами: на пустую клетку и на занятую. В последнем случае следует либо «срубить» фигуру, если она противоположного цвета, либо запретить ход, если он сделан на фигуру своего цвета.

В-седьмых, следует реализовать обработку ситуации «Шах», а при полном рассмотрении также обработку ситуаций «Мат» и «Пат».

Все остальное зависит от конкретной реализации. По желанию программиста и от конкретного проектирования к этому списку может быть добавлен ряд дополнительных свойств и полей. Типы всех полей, используемых в классе, определяются только программистом и могут быть самыми различными. Например, координаты фигур X, Y можно записать как двумя полями типа *byte*, так и использавать стандарный тип *TPoint* или создать собственную запись, состоящую из двух собственных типов и т.д. Вариантов множество, поэтому далее для обозначения всех типов будут использовны условные обозначения. Исходя из вышеописанных рассуждений, класс *Фигура* в общем виде можно описать так:

```
//---------- Общее описание класса TFigura --------
TFigura = class(TРодитель)
 private
 FPlace: TПоложение;
 FColor: TЦвет; //Можно использовать тип Boolean
 procedure SetPlace(Значение: TПоложение);
 //Метод перемещения фигуры на новое место
 procedure SetColor(Значение: TЦвет);
 //Метод смены цвета фигуры (необязателен, если Boolean)
 property MESTO: TПоложение read FPlace write SetPlace;
 //Свойство, отвечающее за положение фигуры на доске
 property Color: TЦвет read FColor write SetColor;
 //Свойство, отвечающее за цвет фигуры
 //(необязательно, если Boolean)
 function HOD(Координаты: TПоложение): boolean; virtual; abstract;
 //Метод, определяющий правило хода фигуры
 function JUMP(Координаты: TПоложение): boolean;
 //Метод проверки на отсутствие занятых клеток на пути хода
 constructor ConstructorName(Координаты: TПоложение; Цвет: TЦвет);
 //Конструктор фигуры
 end;
//-----------------------------------------------------
```

Далее, создадим шесть классов-потомков (подклассов). Для каждого типа фигур – свой подкласс, в котором переопределен метод *HOD*, где описывается правило хода для каждой конкретной фигуры. При правильном ходе фигуры функция *HOD* получает значение *true*. Пример реализации для пешки:

```
TPawn = class(TFigura) //Пешка
 private
 function HOD(XY: TПоложение): boolean; override;
 end;
```

Шаг третий – разработка методов класса TFigura

Нам понадобятся следующие глобальные переменные: массив из тридцати двух элементов для хранения всех фигур и логическая переменная для определения очередности хода, в которой будем хранить цвет фигур, чья очередь хода. Кроме того, нужен счетчик ходов. Опишем их в разделе объявления глобальных переменных следующим образом:

```
Var Figures: array[1..32] of TFigura;
 //Массив фигур
OCHERED: boolean; //Очередность хода
XODOV: byte=0; //Счетчик ходов
```

Конструктор

Теперь переходим к описанию методов. Самый важный из них – конструктор, главной задачей которого является выделение памяти под объект. Непосредственной работой с памятью занимается среда разработки, то есть сама среда Delphi, и к обязанностям программиста относится просто вызов необходимых конструкторов для нужных объектов [13; 14]. Также в конструкторе определяются начальные значения для полей класса и изменяются значения необходимых свойств у создаваемых объектов.

Картинки с изображением фигур будут загружаться из специальной папки с набором нужных изображений или предварительно созданного файла ресурса. Названия картинок нужно строить по определенным правилам: *цвет_ИмяКласса*, например: *black_TPawn*, *white_TKing* и т.д. Для использования файла-ресурса, в котором хранятся изображения фигур, такой файл следует подключить специальной директивой *{$R Фигуры.res}*.

Ниже приведен пример возможного общего вида конструктора для класса *TFigura*, порожденного от класса *TImage*.

```
//------- Общий вид конструктора фигуры -------
constructor
TFigura.ConstructorName(Координаты: TПоложение,
 Цвет: boolean);
 begin
 inherited Create(Form1); //Директива inherited используется
 Parent:=Form1; //в зависимости от реализации
//Указываем объект-контейнер, на который помещается создаваемый объект
//в нашем случае - это форма (или объект класса TKletka,
//если доска состоит из таких объектов)
 FPlace:=Координаты; //Положение фигуры на доске
 MESTO:=FPlace; //Сохранение координат в свойстве для работы
 FColor:=Цвет; //Цвет фигуры
//-------- НАСТРОЙКА IMAGE (ФИГУРЫ) ---------
 Transparent:=true; //Прозрачность фона картинки
 Width:=40; //Высота картинки
 Height:=40; //Ширина картинки
```

```
//--Пример загрузки картинок из ресурса -
if Цвет then
 Picture.Bitmap.Bitmap.LoadFromResource
Name(HInstance,'white_'
 + Self.ClassName+'.bmp');
 else
 Picture. Bitmap.LoadFromResourceName
(HInstance,'black_'
 + Self.ClassName+'.bmp');
//-------------------------------------------------------
 DragMode:=dmAutomatic;    //Автоматическая
обработка события Drag&Drop
 OnDragOver:=Form1.OnImageOver;
 OnDragDrop:=Form1.OnImageDrop;
end;
//-------------------------------------------------------
```

Так как при вызове конструктора объект только создается, то для дальнейшей работы с ним этот объект нужно поместить в компонент массива *Figures[i]* соответствующего типа. Например, поместим белую пешку в качестве первого элемента нашего массива на доску в клетку [1; 2]:

```
Figures[1]:=TPawn.Create(1,2,true);
```

Разумеется, в зависимости от типа координат, входные параметры могут отличаться: здесь приведен только общий пример.

В данной записи (создание пешки) вызывается конструктор класса *TPawn*. Так как конструктор для этого класса в нашем случае не реализован, то будет вызываться конструктор ближайшего родителя, то есть класса *TFigura*, в котором создается картинка, в нее в соответствии с цветом фигуры загружается нужное изображение и помещается на доску по указанным координатам. Для последующего обращения к этой пешке, она сохраняется в компоненте массива *Figures[1]*. Теперь к полям, методам и свойствам этого объекта можно обращаться следующим образом: *Figures[1].ИмяМетода;*

Например, переместить фигуру на другую клетку можно следующим образом: *Figures[1].HOD(Координаты);*

Описание методов записи в свойства здесь приводить не будем, так как их содержание может очень сильно различаться, уточним лишь то, что там необходимо присвоить вводимые координаты в поле положения фигуры, и перемещать в них картинку. Они будут рассмотрены далее. Будем считать, что для исходной шахматной позиции фигуры расставлены.

Метод HOD (Правило хода фигуры)

Следующее, что необходимо сделать – это научить фигуры правильно ходить и не перепрыгивать друг через друга. Необходимо написать эти методы. Правило хода фигуры – полиморфный метод, то есть каждый подкласс описывает его по-своему, поэтому в классе *TFigura* мы объявили его как виртуальный метод (*virtual*). Добавленное слово *abstract* означает, что этот метод используется только в классах-потомках (подклассах), и в основном классе его реализация отсутствует.

Наш метод проверки хода должен возвращать *true*, если ход возможен и *false*, если он не по правилам. В качестве параметров, мы должны передавать в наш метод координаты нового места хода (конечной клетки), так как начальные координаты можно взять из свойства, в котором они хранятся. Другими словами, это будет функция логического типа с одним входным параметром типа *ТПоложение*, в котором будем передавать конечные координаты.

Рассмотрим создание метода правила хода на примере фигуры «Ферзь», который имеет право ходить по вертикалям, горизонталям и диагоналям. Представим конечную координату как *Ferz_XY*. Чтобы можно было понять, правильно ли был выполнен ход, необходимо сравнить координаты по следующим признакам:

- если ход произведен по вертикали, то у начальной и конечной координат совпадает координата X;
- если ход произведен по горизонтали, то у начальной и конечной координат совпадает координата Y;
- если ход произведен по диагонали, то модуль изменения координат по X равен модулю изменения координат по Y.

На языке Object Pascal это можно описать следующим образом:

```
function TQueen.HOD(Ferz_XY:
ТПоложение): boolean;
 begin
 Result:=((MESTO.X=FERZ_XY.X)
or (MESTO.Y=FERZ_XY.Y) or
 (abs(MESTO.X-FERZ_XY.X)=
abs(MESTO.Y-FERZ_XY.Y)));
 end;
```

В этом фрагменте MESTO.X и MESTO.Y отражают начальные, а FERZ_XY.X и FERZ_XY.Y конечные координаты фигуры соответственно. Функция дает значение true, если ход выполнен верно и false - в противном случае.

Исходя из данного примера, несложно написать проверки для фигур: Ладья, Слон и Король. Конь ходит на две клетки по вертикали и на одну по горизонтали или наоборот: две по горизонтали и одна по вертикали. Оба этих случая необходимо соединить логическим оператором «или».

Самой сложной в написании этого метода является пешка. Пешки ходят только вперед, то есть белая пешка ходит только вверх, а черная – только вниз и только на одну клетку. Если же они находятся на «своей» горизонтали (седьмой для черных и второй для белых), то сходить можно на

две клетки; ходят они только прямо, а рубят только по диагонали. Это все надо учесть в реализации хода пешки.

Метод JUMP (Проверка свободного пути)

Для того чтобы фигуры могли правильно ходить, необходимо еще выполнить проверку на наличие занятых клеток на пути хода фигуры, и в случае положительного результата запрещать ход.

Данный метод, аналогично предыдущему, должен возвращать логическое значение: *true* – если ход возможен, *false* – в противном случае. Как и в предыдущем случае, метод будет принимать значение конечной координаты, а начальную – брать из свойства. Рассмотрим, как будет работать метод. Он должен найти фигуру, которая расположена между начальной и конечной клетками движения. Если такая «фигура-барьер» не найдена, то ход разрешен. Выполнять проверки необходимо в трех случаях:

- ход совершен по горизонтали;
- ход совершен по вертикали;
- ход совершен по диагонали.

Под эти три случая не попадает только Конь, а для него такая проверка не требуется, так как он может прыгать через другие фигуры. Все эти случаи описываются совершенно одинаково, поэтому рассмотрим только один из них – вертикаль.

```
function TFigura.JUMP(Координаты: ТПоложение): boolean;
var i: byte;
 begin
 Result:=true; //Ход разрешен
 for i:=1 to 32 do //Проверка всех фигур
 if Figures[i]<> nil then //если фигура есть на доске
 begin
 //------------ Проверка по вертикали ----------
 //ищем фигуру-барьер на пути хода,
 //координата X которой совпадает с нашей
 if (Координаты.X = MESTO.X) and
 (Figures[i].MESTO.X = MESTO.X) and
 //если такая фигура найдена, то проверяем
 //находится ли она на промежутке движения нашей фигуры
 (((Figures[i].MESTO.Y > MESTO.Y) and
 (Figures[i].MESTO.Y < Координаты.Y)) or
 ((Figures[i].MESTO.Y < MESTO.Y) and
 (Figures[i].MESTO.Y > Координаты.Y)))
 then Result:=false;
 //если и это условие выполнено,
 //то занятая клетка найдена – ход запрещен
 end;
```

Аналогичным образом проверяются горизонталь и диагональ. Три таких блока дают нам рабочий метод проверки свободного пути.

Метод SETPLACE (Перемещение фигуры на новое место)

Данный метод производит изменение положения фигуры на доске и существенно зависит от конкретной реализации. Ниже приведены фрагменты кода для случаев конструирования доски с помощью *TPanel* и *TShape:*

```
procedure TFigura.SetPlace(NEW_XY: ТПоложение);
 begin
 FPlace:=NEW_XY;
 Parent:=Form1.DOSKA[NEW_XY.X, NEW_XY.Y];
//для TPanel
 LEFT:=DOSKA[NEW_XY.X, NEW_XY.Y].Left-5;
//для TShape(Столбец)
 TOP:= DOSKA[NEW_XY.X, NEW_XY.Y].Top-5;
//для TShape(Строка)
 end;
```

Метод SHAX (Проверка на шах)

Метод должен проверить каждую фигуру, которая в данный момент присутствует на доске и, если ее цвет не совпадает с цветом атакуемого Короля, узнать, выполняются ли для нее условия методов *HOD* и *JUMP* одновременно на «вражеском королевском месте». Метод *SHAX* должен вернуть в качестве результата значение *true,* если оба метода одновременно вернули значение *true.*

```
function SHAX(Color: TColor): boolean;
var i,King: byte;
 begin
 Result:=false; //Нет шаха
 for i:=1 to 32 do //Автопоиск необходимого короля
 if (Figures[i] is TKing) and (Figures[i].FColor<>Color)
 then King:=i; //Определили номер атакуемого Короля
 //Проверка выполняемости всех описанных выше условий
 for i:=1 to 32 do
 if Figures[i]<> nil then
 if (Figures[i].FColor<>Figures[King].FColor)
 and Figures[i].HOD(Figures[King].MESTO)
 and Figures[i].JUMP(Figures[King].MESTO) then
 begin
 ShowMessage('ШАХ!');
 Result:=true; //Есть шах
 end;
 end;
```

Итак, мы реализовали метод, организующий обработку ситуации «Шах». Его достаточно просто можно доработать таким образом, чтобы в сообщении указывалось, какому именно Королю (белому или черному) объявлен «Шах».

Методы перемещения фигур по доске Drag&Drop

Теперь все подготовлено к тому, чтобы заставить наши фигуры правильно перемещаться по шахматной доске. Создадим соответствующие методы для перетаскивания фигур *Drag&Drop*, тем более что они уже были использованы нами в конструкторе, но их реализация пока отсутствует.

В нашем проекте изначально отсутствуют объекты типа *TImage*, в которых хранятся изображения шахматных фигур, поскольку они создаются в процессе работы программы динамически. А ведь именно их и предстоит претаскивать! Как же создать метод для еще несуществующего объекта?

Для того чтобы создать метод какого-либо события, необходимо использовать обработчик такого события. Для этого необходимо вызвать нужное событие из Инспектора Объектов у данного объекта или у любого другого, у которого есть такое событие. Например, можно найти в Инспекторе Объектов события *OnDragDrop* и *OnDragOver* для формы и щелкнуть дважды мышью в соответствующих пустых полях. Среда программирования Delphi сама создаст пустые обработчики для этих событий. После чего нужно скопировать *списки параметров* появившихся методов в свои собственные методы, которые мы разрабатываем. В нашем примере это методы *OnImageDrop* и *OnImageOver*, которыми необходимо дополнить класс формы (для наглядности в приведенном ниже фрагменте кода они подчеркнуты):

```
TForm1 = class(TForm)
 procedure FormCreate(Sender: TObject);
 procedure OnImageDrop(Sender, Source: TObject; X,Y: Integer);
 procedure OnImageOver(Sender, Source: TObject; X,Y: Integer;
 State: TDragState; var Accept: Boolean);
 private
 { Private declarations }
 public
 { Public declarations }
 end;
```

Для того чтобы эти методы обрабатывали необходимые события, нужно присвоить их этим событиям, как это мы делали ранее при создании доски:

```
DOSKA[i,j].OnDragDrop:=Form1.OnImageDrop;
DOSKA[i,j].OnDragOver:=Form1.OnImageOver;
```

В этом случае ни список параметров, ни скобки уже не указываются. В условных обозначениях это можно еще написать следующим образом:

ЭКЗЕМПЛЯР_ОБЪЕКТА.СОБЫТИЕ:=МЕТОД;

Напомним, что данные методы здесь использованы для случая, когда в качестве родительского класса выбран класс *TImage*. Теперь кратко опишем содержание самих этих методов.

Метод OnImageOver (Переместить объект)

Здесь нужно написать только одну строчку: *Accept:=true;*

Эта запись разрешает использование механизма перетаскивания Drag&Drop, то есть одному объекту (доске) разрешено принять другой объект (фигуру) при отжатии правой клавиши мыши, если один объект находится над другим.

Метод OnDragDrop (Опустить объект)

Здесь нужно описать действия, которые необходимо выполнить после перемещения фигуры на доске на новое место. Как уже говорилось выше, существует два варианта событий: перемещение фигуры на пустую клетку или на занятую. Во втором случае, если клетка занята фигурой соперника, то ее необходимо «срубить», если союзной – запретить перемещение. Ниже представлен сокращенный вариант кода одного из возможных обработчиков событий, записанный в общем виде:

```
//-------------------------------------------------------------
procedure TForm1.OnImageDrop(Sender,Source: TObject; X,Y: Integer);
var NewPlace: TПоложение; //Новые (конечные) координаты фигуры
 OldPlace: TПоложение; //Исходные (начальные) координаты фигуры
 //(используются в случае отмены хода)
 SRUBLENA: byte; //Номер срубленной фигуры в массиве Figures[i]
...
begin
 //Проверка куда совершен ход: на пустую клетку или на занятую
 if Sender is TKletka then
 begin
 //TKletka– имя класса, экземпляром которого является клетка доски
 //далее, вычисляем координаты этой клетки
 NewPlace.X:=((TKletka(Sender).Left -20) div 50)+1;
 NewPlace.Y:=8-((TKletka(Sender).Top -20) div 50);
 //Проверка, возможен ли ход на эту клетку: выполняемость
 //условий методов HOD и JUMP при правильной очередности хода
 if TFigura(Source).HOD(NewPlace) and TFigura(Source).JUMP(NewPlace)
 and (TFigura(Source).FColor=OCHERED) then
 begin
 OldPlace:=TFigura(Source).MESTO;
 TFigura(Source).MESTO:=NewPlace;
 if SHAX(OCHERED) then TFigura(Source).MESTO:=OldPlace;
 //Возврат фигуры на старое место при Шахе
 ...
 end;
 OCHERED:=not OCHERED //Переход хода
 XODOV:=XODOV+1; //Увеличение показаний счетчика ходов
 ...
//-------------------------------------------------------------
```

Остается описать вариант хода с рубкой фигуры противника. Этот случай выявляется после проверки условия *if Sender is TFigura then*, аналогичной проверке хода на пустую клетку, поэтому приводить пример здесь не будем.

Заключение

Изучение объектно-ориентированного программирования дает возможность рассмотреть окружающий мир с двух разных точек зрения: как совокупность объектов и как совокупность процессов. Это позволяет моделировать предметную область решаемой задачи. Для освоения объектно-ориентированного подхода на стадии обучения очень важен подбор наглядных примеров, которые показывают его преимущества и эффективность в реализации конкретных решений. В этом могут помочь примеры по разработке игровых программ, которые пользуются большой популярностью у обучаемых, что в данной статье и было продемонстрировано на примере рассмотрения общих подходов при создании приложения «Шахматы». Для быстрого освоения технологии программирования в среде Delphi можно воспользоваться учебными методическими пособиями, предложенными автором данной статьи [6–8].

ЛИТЕРАТУРА

1. Александровский А. Д. Delphi 4. Шаг в будущее. М. : ДМК, 1999. 528 с.
2. Баас Р., Фервай М., Гюнтер Х. Delphi 5. Лучший инструмент для разработки эффективных и надежных Windows-приложений. Киев : BHV, 2000. 496 с.
3. Бобровский С. И. Delphi 7 : учебный курс. СПб. : Питер, 2003. 736 с.
4. Васильев А. Н. Java. Объектно-ориентированное программирование : учебное пособие. СПб. : Питер, 2011. 400 с.
5. Гофман В. Э., Хомоненко А. Д. Delphi 5. Наиболее полное руководство в подлиннике. СПб. : BHV, 2000. 800 с.
6. Емельянов Д. А., Лапенок М. В. Технология объектно-ориентированного программирования в среде Delphi : учебное пособие. Екатеринбург : Изд-во УрГПУ, 2006. 92 с.
7. Емельянов Д. А. Практикум по решению задач: введение в Delphi : учебное пособие. Екатеринбург : УрГПУ. 2009. 225 с.
8. Емельянов Д. А. Объектно-ориентированное программирование в среде Delphi : учебное пособие. Екатеринбург : УрГЭУ, 2012. 124 с.
9. Культин Н. Б. Программирование на Object Pascal в Delphi 5. СПб. : BHV, 2000. 464 с.
10. Комарова Е. С. Практикум по программированию на языке Паскаль : учебное пособие. М. : Директ-Медиа, 2015. 208 с.
11. Ремнев Н. А., Федотова С. В. Курс Delphi для начинающих. Полигон нестандартных задач. М. : Солон-Пресс, 2014. 360 с.
12. Рубанцов В. Delphi в примерах, играх и программах. СПб. : Наука и техника, 2011. 672 с.
13. Фаронов В. В. Delphi 6: учебный курс. М. : Издатель Молгачева С. В., 2001. 672 с.
14. Фленов М. Е. Библия Delphi. 3-е издание. СПб. : БХВ-Петербург, 2015. 688 с.
15. Федотова С. В. Создание Windows-приложений в среде Delphi. М. : Солон-Пресс, 2004. 410 с.

LITERATURA

1. Aleksandrovskiy A. D. Delphi 4. Shag v budushchee. M. : DMK, 1999. 528 s.
2. Baas R., Fervay M., Gyunter Kh. Delphi 5. Luchshiy instrument dlya razrabotki effektivnykh i nadezhnykh Windows-prilozheniy. Kiev : BHV, 2000. 496 c.
3. Bobrovskiy S. I. Delphi 7 : uchebnyy kurs. SPb. : Piter, 2003. 736 s.
4. Vasil'ev A. N. Java. Ob"ektno-orientirovannoe programmirovanie : uchebnoe posobie. SPb. : Piter, 2011. 400 c.
5. Gofman V. E., Khomonenko A. D. Delphi 5. Naibolee polnoe rukovodstvo v podlinnike. SPb. : BHV, 2000. 800 s.
6. Emel'yanov D. A., Lapenok M. V. Tekhnologiya ob"ektno-orientirovannogo programmirovaniya v srede Delphi : uchebnoe posobie. Ekaterinburg : UrGPU, 2006. 92 s.
7. Emel'yanov D. A. Praktikum po resheniyu zadach: vvedenie v Delphi : uchebnoe posobie. Ekaterinburg : UrGPU. 2009. 225 s.
8. Emel'yanov D. A. Ob"ektno-orientirovannoe programmirovanie v srede Delphi : uchebnoe posobie. Ekaterinburg : UrGEU, 2012. 124 s.
9. Kul'tin N. B. Programmirovanie na Object Pascal v Delphi 5. SPb. : BHV, 2000. 464 c.
10. Komarova E. S. Praktikum po programmirovaniyu na yazyke Paskal' : uchebnoe posobie. M. : Direkt-Media, 2015. 208 c.
11. Remnev N. A., Fedotova S. V. Kurs Delphi dlya nachinayushchikh. Poligon nestandartnykh zadach. M. : Solon-Press, 2014. 360 s.
12. Rubantsov V. Delphi v primerakh, igrakh i programmakh. SPb. : Nauka i tekhnika, 2011. 672 s.
13. Faronov V. V. Delphi 6: uchebnyy kurs. M. : Izdatel' Molgacheva S. V., 2001. 672 s.
14. Flenov M. E. Bibliya Delphi. 3-e izdanie. SPb. : BKhV-Peterburg, 2015. 688 s.
15. Fedotova S. V. Sozdanie Windows-prilozheniy v srede Delphi. M. : Solon-Press, 2004. 410 s.

Статью рекомендует д-р пед. наук, проф. Б. Е. Стариченко